Sobre proyecto final

E. Rivas

Junio de 2013

Introducción

El objetivo de esta presentación es introducir el desarrollo de los trabajos prácticos.

Introducción

El objetivo de esta presentación es introducir el desarrollo de los trabajos prácticos.

En el trabajo final no solo importará que el programa sea correcto, i.e. que haga lo requerido, si no también su forma de presentación.

Introducción

El objetivo de esta presentación es introducir el desarrollo de los trabajos prácticos.

En el trabajo final no solo importará que el programa sea correcto, i.e. que haga lo requerido, si no también su forma de presentación.

Durante el curso los archivos se compilaban a mano, generalmente estábamos enfocados en resolver los problemas de la práctica, y no importaba demasiado la organización.

Estructura de directorios

Los proyectos suelen desarrollarse en una estructura de directorios como:

```
bin/ -- carpeta con los binarios generados
build/ -- carpeta donde se compila el programa
doc/ -- carpeta con documentación
src/ -- carpeta con archivos fuente
LICENSE -- archivo con la licencia
Makefile -- archivo generación
```

Estructura de directorios

Los proyectos suelen desarrollarse en una estructura de directorios como:

```
bin/ -- carpeta con los binarios generados
build/ -- carpeta donde se compila el programa
doc/ -- carpeta con documentación
src/ -- carpeta con archivos fuente
LICENSE -- archivo con la licencia
Makefile -- archivo generación
```

Esta estructura no es fija, y varía de proyecto en proyecto. Una buena manera de empezar a estructurar un proyecto es copiar la estructura de algún programa open source.

Estilo del código

Le llamaremos *estilo del código* a un conjunto de reglas que se seguirán a la hora de escribir código. En particular, nos enfocaremos en la parte "estética" y la indentación.

Estilo del código

Le llamaremos *estilo del código* a un conjunto de reglas que se seguirán a la hora de escribir código. En particular, nos enfocaremos en la parte "estética" y la indentación.

```
void f(int x)
{
  for (i =0;i<5;i++) {
  printf("%d",x);
  }
}
```

```
void f(int x) {
  for (i = 0; i < 5; i++)
    printf("%d", x);
  }
}
```

Estilo del código

Le llamaremos *estilo del código* a un conjunto de reglas que se seguirán a la hora de escribir código. En particular, nos enfocaremos en la parte "estética" y la indentación.

```
void f(int x)
{
  for (i =0;i<5;i++) {
  printf("%d",x);
  }
}
```

```
void f(int x) {
  for (i = 0; i < 5; i++)
    printf("%d", x);
  }
}
```

Habrá que elegir un estilo y utilizarlo consistentemente durante todo el proyecto.
Existen varios estilos clásicos que se pueden usar: K&R, Allman, GNU, BSD.

Indent

El comando `indent` puede ayudarnos a mantener un estilo de código.

```
indent main.c
```

Leerá el archivo `main.c` y lo emprolijará de acuerdo a un coding style elegido.

Indent

El comando `indent` puede ayudarnos a mantener un estilo de código.

```
indent main.c
```

Leerá el archivo `main.c` y lo emprolijará de acuerdo a un coding style elegido.

Según los parámetros, se tomarán diferentes estilos de código.

```
indent -st -bap -bli0 -i4 -l79 -ncs -npcs
       -npsl -fca -lc79 -fc1 -ts4 file.c
```

por ej. sigue el estilo Allman.

Existen formatos predefinidos: `-kr`, `-gnu`, `-orig`, etc.

Assert

La cabecera `assert.h` define una macro `assert`.

Se utiliza poniendo una expresión como argumento. Si esta expresión vale 0, entonces el programa emite un mensaje y aborta.

Assert

La cabecera `assert.h` define una macro `assert`.

Se utiliza poniendo una expresión como argumento. Si esta expresión vale 0, entonces el programa emite un mensaje y aborta.

```
int sum(int n) {
  assert(n > 0);
  return (n*(n+1))/2;
}
```

Assert

La cabecera `assert.h` define una macro `assert`.

Se utiliza poniendo una expresión como argumento. Si esta expresión vale 0, entonces el programa emite un mensaje y aborta.

```
int sum(int n) {
  assert(n > 0);
  return (n*(n+1))/2;
}
```

Generalmente conviene advertir los `assert` en un comentario sobre la función.

Comentarios

Es recomendable comentar el código con la cantidad justa de información. Hay que evitar:

- Escasez: ausencia de comentarios que sean necesarias para el uso de cada función.
- Abundancia: exceso de comentarios que dificultan la lectura del código, explicando obviedades.

Comentarios

Es recomendable comentar el código con la cantidad justa de información. Hay que evitar:

- Escasez: ausencia de comentarios que sean necesarias para el uso de cada función.
- Abundancia: exceso de comentarios que dificultan la lectura del código, explicando obviedades.

```
int i = 0; /* inicializo a i en 0 */

/* f: toma un entero y devuelve otro */
int f(int n) {
  assert(n > 0);
  return floor(n/3);
}
```

Makefile

`make` es una herramienta para la generación de automática de archivos.

Generalmente se utiliza para generar un archivo compilado a partir de su código fuente. `make` llamará a los compiladores (gcc en nuestro caso), de manera de ir construyendo el binario final.

Makefile

`make` es una herramienta para la generación de automática de archivos.

Generalmente se utiliza para generar un archivo compilado a partir de su código fuente. `make` llamará a los compiladores (`gcc` en nuestro caso), de manera de ir construyendo el binario final.

Nuestro objetivo es que al escribir `make` en el directorio del proyecto, de manera automática se compilen todos los archivos necesarios para generar el programa final.

Makefile - ejemplo

```
CC=gcc
CFLAGS=-c -Wall
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
$(CC) $(OBJECTS) -o $@

.c.o:
$(CC) $(CFLAGS) $< -o $@
```

Makefile - uso

```
e@d:~/newproj$ ls
Makefile  factorial.c  hello.c  main.c
e@d:~/newproj$ make
.
.
e@d:~/newproj$ ls
Makefile  factorial.c  factorial.o  hello  hello.c
hello.o   main.c       main.o
e@d:~/newproj$ make clean
.
.
e@d:~/newproj$ ls
Makefile  factorial.c  hello.c  main.c
e@d:~/newproj$
```

Testing

El testing es utilizado para verificar que alguna parte del programa es correcto.

Testing

El testing es utilizado para verificar que alguna parte del programa es correcto.

Si tenemos una función `int add(int x, int y)`, podemos definir una función de testing:

```
void test_add() {
  /* numeros positivos */
  assert(add(3, 4) == 7);
  /* numeros negativos */
  assert(add(-3, -2) == -5);
  /* numeros grandes */
  assert(add(5000, 6001) == 11001);
  /* cero es neutro */
  assert(add(0, 0) == 0);
}
```

Conclusiones

Para cerrar, en el proyecto final deberán:

- Armar una estructura de directorios.
- Seguir un coding style.
- Comentar apropiadamente.
- Testear funciones.
- Escribir un `Makefile`.

Conclusiones

Para cerrar, en el proyecto final deberán:

- Armar una estructura de directorios.
- Seguir un coding style.
- Comentar apropiadamente.
- Testear funciones.
- Escribir un `Makefile`.

Existe gran cantidad de manuales y tutoriales sobre estos tópicos.

Conclusiones

Para cerrar, en el proyecto final deberán:

- Armar una estructura de directorios.
- Seguir un coding style.
- Comentar apropiadamente.
- Testear funciones.
- Escribir un `Makefile`.

Existe gran cantidad de manuales y tutoriales sobre estos tópicos.

Un libro buen libro sobre algunos de estos temas es La práctica de la Programación, por Kernighan y Pike.